DTIC FILE COPY

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AD-A223 597

IT DATE

8. REPORT TYPE AND DATES COVERED
Final 30 Nov. 1989 to 30 Nov. 1990

4. Ada Compiler Validation Summary Report: U.S. Navy AdaVAX, Version 3.0 (/NO_OPTIMIZE Option), VAX 8350 and VAX 11/785 (Host & Target), 891130S1.10209

5. FUNDING NUMBERS

6. AUTHOR(S)
National Institute of Standards and Technology
Gaithersburg, MD
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Institute of Standards and Technology
National Computer Systems Laboratory
Bldg. 255, Rm. A266
Gaithersburg, MD 20899
USA

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

U.S. NAVY, AdaVAX, Version 3.0 (/NO_OPTIMIZE Option), Gaithersburg, MD, VAX 8350 and VAX 11/785 (Host & Target), 891130S1.10209, ACVC 1.10.

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

AVF Control Number: NIST89USN555_1_1.10
DATE VSR COMPLETED BEFORE ON-SITE: 08-11-89
DATE VSR COMPLETED AFTER ON-SITE: 11-29-89
DATE VSR MODIFIED PER AVO COMMENTS: 12-29-89
DATE VSR MODIFIED PER AVO COMMENTS: 04-27-90

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 891130S1.10209
U.S. NAVY
AdaVAX, Version 3.0 (/NO_OPTIMIZE Option)
VAX 8350 and VAX 11/785 Hosts and VAX 8350 and VAX 11/785 Target

Completion of On-Site Testing:
November 30, 1989

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

AVF Control Number: NIST89USN555_1_1.10
DATE COMPLETED BEFORE ON-SITE: 08-11-89
DATE COMPLETED AFTER ON-SITE: 11-29-89

Ada Compiler Validation Summary Report:

Compiler Name: AdaVAX, Version 3.0 (/NO_OPTIMIZE Option)

Certificate Number: 891130S1.10209

Hosts: VAX 8350 and VAX 11/785 under VMS, Version 5.1

Target: VAX 8350 and VAX 11/785 under VMS, Version 5.1

Testing Completed November 30, 1989 Using ACVC 1.10

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group
Engineering Division
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report. The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

On-site testing was completed November 30, 1989 at Newport, Rhode Island.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be

directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the Commentary point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn	An ACVC test found to be incorrect and not used to check test conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class

A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure

CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: AdaVAX, Version 3.0 (/NO_OPTIMIZE Option)

ACVC Version: 1.10

Certificate Number: 891130S1.10209

Host Computers:

Machine: VAX 8350 and VAX 11/785

Operating System: VMS, Version 5.1

Memory Size: 16MBytes / 16MBytes

Target Computer:

Machine: VAX 8350 and VAX 11/785

Operating System: VMS, Version 5.1

Memory Size: 16MBytes / 16MBytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types LONG_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) All of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision

and uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised for pre-defined integer comparison and for pre-defined integer membership. NO EXCEPTION is raised for large_int comparison or for large_int membership. NUMERIC_ERROR is raised for small_int comparison and for small_int membership when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is not gradual. (See tests C45524A..E (5 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round away from zero. (See tests C46012A..E (5 tests).)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..E (5 tests).)

(3) The method used for rounding to integer in static universal real expressions is round toward zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

(2) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to an array

type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises no exception. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) All choices were not evaluated before CONSTRAINT_ERROR is raised when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is supported for functions or procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

(3) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(4) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

(5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

(6) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

(7) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

(8) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

(9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output. (Note that for this implementation, all SEQUENTIAL_IO and DIRECT_IO support was tested on a disk; for both SEQUENTIAL_IO and DIRECT_IO, this implementation supports both disk and tape.)

(1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) USE_ERROR is raised when Mode IN_FILE is not supported for the operation of CREATE for SEQUENTIAL_IO. (See test CE2102D.)

(4) USE_ERROR is raised when Mode IN_FILE is not supported for the operation of CREATE for DIRECT_IO. (See test CE2102I.)

(5) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102J, CE2102R, CE2102T, and CE2102V.)

(6) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102I..K (3 tests).)

(7) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(8) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(9) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

(10) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

(11) Temporary sequential files are given names and not deleted when closed. (See test CE2108A.)

(12) Temporary direct files are given names and not deleted when closed. (See test CE2108C.)

(13) Temporary text files are given names and not deleted when closed. (See test CE3112A.)

(14) More than one internal file can be associated with each external file for sequential files when reading only. (See test CE2107A and CE2102L.)

(15) Only one internal file can be associated with each external file for sequential files when writing. (See tests CE2107B..E (4 tests), CE2110B, and CE2111D.)

(16) More than one internal file can be associated with each external file for direct files when reading. (See test CE2107F.)

(17) Only one internal file can be associated with each external file for direct files when writing. (See tests CE2107G..H (2 tests), CE2110D and CE2111H.)

(18) More than one internal file can be associated with each external file for text files when reading only. (See CE3111A.)

(19) More than one internal file can be associated with each external file for text files when reading or writing. (See tests CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 519 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 37 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	127	1132	1809	17	23	46	3154
Inapplicable	2	6	506	0	5	0	519
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	192	543	496	245	172	99	161	331	137	36	252	212	278	3154
Inapplicable	20	106	184	3	0	0	5	1	0	0	0	157	43	519
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

A39005G	B97102E	C97116A	BC3009B	CD2A62D	CD2A63A
CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B	CD2A66C
CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D	CD2A76A
CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G	CD2A84M
CD2A84N	CD2B15C	CD2D11B	CD5007B	CD5011O	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B	E28005C	ED7004B	ED7005C	ED7005D
ED7006C	ED7006D				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 519 tests were inapplicable for the reasons indicated:

a. The following 285 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)

C45241F..Y (20 tests) C45321F..Y (20 tests)
C45421F..Y (20 tests) C45521F..Z (21 tests)
C45524F..Z (21 tests) C45621F..Z (21 tests)
C45641F..Y (20 tests) C46012F..Z (21 tests)

b. C35508I, C35508J, C35508M, and C35508N are not applicable because they include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

c. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

d. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

C45231B C45304B C45502B C45503B C45504B
C45504E C45611B C45613B C45614B C45631B
C45632B B52004E C55B07B B55B09D B86001V
CD7101E

e. C45531M..P (4 tests), C45532M..P (4 tests) are not applicable because this implementation does not support a 48 bit integer machine size.

f. B86001X, C45231D, and CD7101G (3 tests) are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

g. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

h. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

i. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

j. CD1009C, CD2A41A, CD2A41B, CD2A41E, CD2A42A, CD2A42B, CD2A42C, CD2A42D, CD2A42E, CD2A42F, CD2A42G, CD2A42H, CD2A42I, CD2A42J (14 tests) are not applicable because this implementation does not support 'SIZE representations for floating-point types.

k. CD1C04C is not applicable because this implementation does not support 'SMALL specification clause for a derived fixed point type when it is inherited from the parent.

l. CD2A51A, CD2A51B, CD2A51C, CD2A51D, CD2A51E, CD2A52A, CD2A52B,

CD2A52C, CD2A52D, CD2A52G, CD2A52H, CD2A52I, CD2A52J, CD2A53A, CD2A53B, CD2A53C, CD2A53D, CD2A53E, CD2A54A, CD2A54B, CD2A54C, CD2A54D, CD2A54G, CD2A54H, CD2A54I, CD2A54J, ED2A56A (27 tests) are not applicable because this implementation does not support 'SIZE representations for fixed-point types.

m. CD2A61A, CD2A61B, CD2A61C, CD2A61D, CD2A61F, CD2A61H, CD2A61I, CD2A61J, CD2A61K, CD2A61L, CD2A62A, CD2A62B CD2A62C (13 tests) are not applicable because this implementation does not support size specifications for array types that imply compression of component storage.

n. CD2A71A, CD2A71B, CD2A71C, CD2A71D, CD2A72A, CD2A72B, CD2A72C, CD2A72D (8 tests) are not applicable because this implementation does not support the 'SIZE specification for record types implying compression of component storage.

o. CD2A84B, CD2A84C, CD2A84D, CD2A84E, CD2A84F, CD2A84G, CD2A84H, CD2A84I, CD2A84K, CD2A84L (10 tests) are not applicable because this implementation does not support 'SIZE representations for access types.

p. CD5003B, CD5003C, CD5003D, CD5003E, CD5003F, CD5003G, CD5003H, CD5003I, CD5011A, CD5011C, CD5011E, CD5011G, CD5011I, CD5011K, CD5011M, CD5011Q, CD5012A, CD5012B, CD5012E, CD5012F, CD5012I, CD5012J, CD5012M, CD5013A, CD5013C, CD5013E, CD5013G, CD5013I, CD5013K, CD5013M, CD5013O, CD5013S, CD5014A, CD5014C, CD5014E, CD5014G, CD5014I, CD5014K, CD5014M, CD5014O, CD5014S, CD5014T, CD5014V, CD5014X, CD5014Y, CD5014Z (46 tests) are not applicable because this implementation does not support 'ADDRESS clauses for variables.

q. CD5011B, CD5011D, CD5011F, CD5011H, CD5011L, CD5011N, CD5011R, CD5011S, CD5012C, CD5012D, CD5012G, CD5012H, CD5012L, CD5013B, CD5013D, CD5013F, CD5013H, CD5013L, CD5013N, CD5013R, CD5014B, CD5014D, CD5014F, CD5014H, CD5014J, CD5014L, CD5014N, CD5014R, CD5014U, CD5014W (30 tests) are not applicable because this implementation does not support 'ADDRESS clauses for constants.

r. CD2A91A, CD2A91B, CD2A91C, CD2A91D, CD2A91E (5 tests) are not applicable because this implementation does not support the 'SIZE representation clauses for task types.

s. AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

t. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

u. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

v. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

w. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

x. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

y. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

z. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

aa. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

ab. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

ac. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ad. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ae. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

af. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ag. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ah. CE2105A is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for SEQUENTIAL_IO.

ai. CE2105B is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for DIRECT_IO.

aj. CE2107B..E (4 tests), CE2107L, CE2110B CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is reading or writing for sequential files. The proper exception is raised when multiple access is attempted.

ak. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

al. CE3102F is inapplicable because text file RESET is supported by this implementation.

am. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

an. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

ao. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

ap. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.

aq. CE3109A is inapplicable because text file CREATE with IN_FILE mode is not supported by this implementation.

ar. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 37 tests.

CC3126A was modified by inserting the initializing expression ":= (others => 'H')" into line numbered 117. With this modification, this test reports PASS.

For this implementation CD2C11A and CD2C11B were modified by inserting the initialization ":= 5.0" into variable W's declaration (note that W is declared along with one or two other variables in a single object

declaration; the initialization is not needed for them, but does not affect their use). With these modification, these tests report PASS.

The following 34 tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B28003A	B28003C	B2A003A	B33201C	B33202C	B33203C	B33301B
B37106A	B37201A	B37301I	B38003A	B38003B	B38009A	B38009B
B44001A	B44004A	B51001A	B54A01L	B91001H	B95063A	BB1006B
BC1002A	BC1102A	BC1109A	BC1109B	BC1109C	BC1109D	BC1201F
BC1201G	BC1201H	BC1201I	BC1201J	BC1201L	BC3013A	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the AdaVAX, Version 3.0 (/NO_OPTIMIZE Option) compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the AdaVAX, Version 3.0 (/NO_OPTIMIZE Option) compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computers:	VAX 8350 and VAX 11/785
Host operating system:	VMS, Version 5.1
Target computers:	VAX 8350 and VAX 11/785
Target operating system:	VMS, Version 5.1
Compiler:	AdaVAX, Version 3.0 (/NO_OPTIMIZE Option)
Linker:	LNKVAX Version 3.0
Importer:	IMPVAX Version 3.0
Exporter:	EXPVMS Version 3.0
Runtime System:	RSL Version 3.0

A magnetic tape containing all tests except for withdrawn tests was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were modified on-site.

The contents of the magnetic tape were loaded directly onto the host computers.

The ACVC Version 1.10 was compiled, linked, and all executable tests were run on the host/target VAX 8350 as listed above. Results were printed from the VAX 8350.

The ACVC Version 1.10 was compiled, linked, and all executable tests were run on the host/target VAX 11/785 as listed above. Results were printed from the VAX 11/785 computer.

The compiler was tested using command scripts provided by U.S. NAVY and reviewed by the validation team. See Appendix E for a complete listing of the compiler options for this implementation. The compiler options invoked during this test were:

For A, C, D, L Tests:

/SUMMARY /NO_TRACE_BACK /NO_OPTIMIZE /NO_LIST

For B, E Tests:

/SUMMARY /NO_TRACE_BACK /NO_OPTIMIZE /SOURCE /NO_LIST

Unless explicitly stated the following are the default options:

NO_SOURCE, NO_MACHINE, NO_ATTRIBUTE, NO_CROSS_REFERENCE, NO_DIAGNOSTICS, NO_SUMMARY, NO_NOTES, PRIVATE, CONTAINER_GENERATION, CODE_ON_WARNING, LIST, NO_MEASURE, DEBUG, TRACE_BACK, NO_OPTIMIZE, CHECKS

Tests were compiled, linked, and executed as appropriate using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings were examined on-site by the validation team.

3.7.3 Test Site

Testing was conducted at Newport, Rhode Island and was completed on November 30, 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

U.S. NAVY has submitted the following Declaration of Conformance concerning the AdaVAX, Version 3.0 (/NO_OPTIMIZE Option).

DECLARATION OF CONFORMANCE

Customer: U.S. NAVY

Ada Validation Facility:

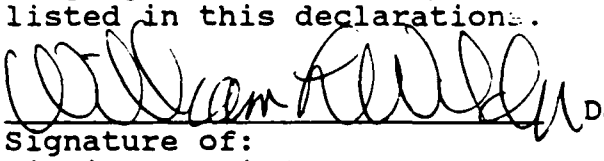
Ada Validation Facility
National Computer Systems Laboratory (NCSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Compiler Validation Capability (ACVC) Version: 1.10

Ada Implementation: AdaVAX, Version 3.0 (/NO_OPTIMIZE Option)
Host Computer Systems: VAX 8350 and VAX 11/785
Host OS and Version: VMS, Version 5.1
Target Computer System: VAX 8350 and VAX 11/785
Target OS and Version: VMS, Version 5.1

Customer's Declaration

I, the undersigned, representing U.S. NAVY, declare that the U.S. NAVY has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.



Date: 10/12/89

Signature of:
William L. Wilder,
U.S. NAVY

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the AdaVAX, Version 3.0 (/NO_OPTIMIZE Option) compiler, as described in this Appendix, are provided by U.S. NAVY. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

...

 type INTEGER is range -32_768 .. 32_767;
 type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

 type FLOAT is digits 6 range
  - (2#0.1111_1111_1111_1111_1111_1#E127) ..
    (2#0.1111_1111_1111_1111_1111_1#E127);
 type LONG_FLOAT is digits 9 range
  - (2#0.1111_1111_1111_1111_1111_1111_1111_111#E127) ..
    (2#0.1111_1111_1111_1111_1111_1111_1111_111#E127);

 type DURATION is delta 2.0 ** (-14) range
    -131_072.0 .. 131_072.0 -2.0 ** (-14);

...

end STANDARD;
```

Section 6

The Ada Language for the VAX Target

The source language accepted by the compiler is Ada, as described in the Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A-1983, 17 February 1983 ("Ada Language Reference Manual").

The Ada definition permits certain implementation dependencies. Each Ada implementation is required to supply a complete description of its dependencies, to be thought of as Appendix F to the Ada Reference Manual. This section is that description for the VAX/VMS target.

6.1 Options

There are several compiler options provided by all ALS/N Compilers that directly affect the pragmas defined in the Ada Reference Manual. These compiler options currently include the CHECKS and OPTIMIZE options which affect the SUPPRESS and OPTIMIZE pragmas, respectively. A complete list of ALS/N Compiler options can be found in Section 10.2.

The CHECKS option enables all run-time error checking for the source file being compiled, which can contain one or more compilation units. This allows the SUPPRESS pragma to be used in suppressing the run-time checks discussed in the Ada Reference Manual, but note that the SUPPRESS pragma(s) must be applied to each compilation unit. The NO_CHECKS option disables all run-time error checking for all compilation units within the source file and is equivalent to SUPPRESSing all run-time checks within every compilation unit.

The OPTIMIZE option enables all compile-time optimizations for the source file being compiled, which can contain one or more compilation units. This allows the OPTIMIZE pragma to request either TIME-oriented or SPACE-oriented optimizations be perfromed, but note that the OPTIMIZE pragma must be applied to each compilation unit. If the OPTIMIZE pragma is not present, the ALS/N Compiler's Global Optimizer tends to optimize for TIME over SPACE. The NO_OPTIMIZE option disables all compile-time optimizations for all compilation units within the source file regardless of whether or not the OPTIMIZE pragma is present.

6.2 Pragmas

Both implementation-defined and Ada language-defined pragmas are provided by all ALS/N Compilers. The syntax defined in the Ada Reference Manual allows pragmas as the only element in a compilation, before a compilation unit, at defined places within a compilation unit, or following a compilation unit. The ALS/N Compilers associates pragmas with compilation units as follows:

a. If a pragma appears before any compilation unit in a compilation, it will affect all following compilation units, as specified below, and in the Ada Reference Manual.

b. If a pragma appears inside a compilation unit, it will be associated with that compilation unit, and in listings associated with that compilation unit as described in the Ada Reference Manual, or in this document.

c. If a pragma follows a compilation unit, it will be associated with the preceding compilation unit, and the effects of the pragma will be found in the container of that compilation unit, and in listings associated with that container.

The pragmas MEMORY_SIZE, STORAGE_UNIT and SYSTEM_NAME are described in Section 13.7 of the Ada Language Reference Manual. They may appear only at the start of the first compilation when creating a new program library. In the ALS/N, however, since program libraries are created by the Program Library Manager and not by the compiler, the use of these pragmas is obviated. If they appear anywhere, a diagnostic of severity level WARNING is generated.

6.2.1 Language-defined Pragmas

The following notes specify the language-required definitions of the predefined pragmas. Unmentioned language-defined pragmas are implemented as defined by the Ada Language Reference Manual.

pragma INLINE (subprogram_name);

There are three instances in which the INLINE pragma is ignored. Each of these cases produces a warning message which states that the INLINE did not occur.

a. If a call to an INLINEd subprogram is compiled before the actual body of the subprogram has been compiled (a routine call is made instead).

b. If the INLINEd subprograms compilation unit depends on the compilation unit of its caller (a routine call is made instead).

c. If an immediately recursive subprogram call is made within the body of the INLINEd subprogram (the pragma INLINE is ignored entirely).

pragma INTERFACE (language_name, subprogram_name);

Two language_names will be recognized and implemented:

ASMVAX_JSB, and ASMVAX_CALLS.

The language_name ASMVAX_JSB indicates that a subprogram written in the VAX/VMS assembler language will be called with a JSB instruction and the parameters passed in registers. The language_name ASMVAX_CALLS will provide interface to a VAX assembler language subprogram via the CALLS instructions, with the parameters passed on the stack, with the same parameter passing conventions used for calling Ada subprograms.

The users must ensure that an assembly-language body container for this specification will exist in the program library before linking.

pragma PAGE;

This is a listing control pragma using. Using the pragma causes a page break in the listing; all code that follows pragma PAGE will start on a new page. The pragma may be placed anywhere inside a compilation unit.

pragma OPTIMIZE (arg)

This pragma is effective only when the "OPTIMIZE" option has been given to the compiler. The argument is either TIME or SPACE. If TIME is specified, the optimizer concentrates on optimizing code execution time. If SPACE is specified, the optimizer concentrates on optimizing code size.

pragma PRIORITY (arg)

The PRIORITY argument is an integer static expression value of predefined integer subtype PRIORITY. The pragma has no effect in a location other than a task (type) specification or outermost declarative part of a subprogram. If the pragma appears in the declarative part of a subprogram, it has no effect unless that subprogram is designated as the "main" subprogram at link time.

pragma SUPPRESS (arg[,arg])

Pragmas to suppress OVERFLOW_CHECK will have no effect for operations of integer types.

A SUPPRESS pragma will have effect only within the compilation unit in which it appears, except that a SUPPRESS of ELABORATION_CHECK applied at the declaration of a subprogram or task unit will apply to all calls or activations.

6.2.2 Implementation-defined Pragmas

The following item is the only implementation-defined pragma:

pragma TITLE (arg);

This is a listing control pragma. It specifies a character string that is to appear on the second line of each page of every listing produced for a compilation unit in the compilation. This title is printed until a new pragma TITLE is encountered. The pragma should be placed after the compilation unit specification; if it is not, a warning message is issued. The argument is a string literal.

6.2.3 Scope of Pragmas

The scope of pragmas is as described in the Ada Reference Manual except as noted below.

MEMORY_SIZE No scope, but a WARNING diagnostic is generated

PAGE No scope

STORAGE_SIZE No scope, but a WARNING diagnostic is generated

SYSTEM_NAME No scope, but a WARNING diagnostic is generated

TITLE No scope

6.3 Attributes

There is one implementation-defined attribute in addition to the predefined attributes found in Appendix A of the Ada Reference Manual.

X'DISP

A value of type UNIVERSAL_INTEGER which corresponds to the displacement that is used to address the first storage unit occupied by a data object X at a static offset within an implemented activation record.

This attribute differs from the ADDRESS attribute in that ADDRESS supplies the absolute address while DISP supplies the displacement relative to some base value (such as a stack frame pointer). It is the user's responsibility to determine the base value relevant to the attribute.

The following notes augment the language-required definitions of the predefined attributes found in Appendix A of the Ada Reference Manual.

T'MACHINE_ROUNDS	is false.
T'MACHINE_RADIX	is 2.
T'MACHINE_MANTISSA	if the size of the base type T is 32, MACHINE_MANTISSA is 24. if the size of the base type T is 64, MACHINE_MANTISSA is 56.
T'MACHINE_EMAX	is 127
T'MACHINE_EMIN	is -127
T'MACHINE_OVERFLOWS	is true.

6.4 Predefined Language Environment

The predefined Ada language environment consists of the packages STANDARD and SYSTEM described below.

6.4.1 Package

The Package STANDARD contains the following definitions in addition to those specified in Appendix C of the Ada Language Reference Manual:

```
-- For this implementation, there is no corresponding body.

-- The universal type UNIVERSAL_INTEGER is predefined for Ada.

type INTEGER is range -32_768 ..  32_767;

type LONG_INTEGER is range -2_147_483_648 ..  2_147_483_647;

-- The universal type UNIVERSAL_REAL is predefined for Ada.

type FLOAT is digits 6 range
    - (2#0.1111_1111_1111_1111_1111_1#E127) ..
      (2#0.1111_1111_1111_1111_1111_1#E127);

type LONG_FLOAT is digits 9 range
     -(2#0.1111_1111_1111_1111_1111_1111_1111_111#E127) ..
      (2#0.1111_1111_1111_1111_1111_1111_1111_111#E127);
-- Predefined subtypes within the Ada Language:

subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST; -- 32_767
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST; -- 32_767
subtype LONG_NATURAL  is LONG_INTEGER
                         range 0 .. LONG_INTEGER'LAST;
subtype LONG_POSITIVE is LONG_INTEGER
                         range 1 .. LONG_INTEGER'LAST;

-- Predefined STRING type within the Ada Language:

type STRING is array (POSITIVE range <>) of CHARACTER;
pragma PACK(STRING);

-- The type DURATION is predefined for use with Ada DELAY.

type DURATION is delta 2.0 ** (-14)
         range -131_072.0 .. 131_072.0  -2.0 ** (-14);

-- The predefined operators for the type DURATION are the same
-- as for any fixed point type within the Ada language.
```

6.4.2 Package SYSTEM

Within the various implementations, no corresponding package body is required for the package SYSTEM. The package SYSTEM is as follows;

```
package SYSTEM is

   type ADDRESS is new LONG_INTEGER;
   type NAME     is (AdaVAX, Ada_L, Ada_M);
   SYSTEM_NAME   : constant NAME := AdaVAX;
   STORAGE_UNIT : constant :=  8;
   MEMORY_SIZE   : constant :=  2**30 - 1;

-- System-Dependent Named Numbers:

   MIN_INT       : constant :=  -(2**31);
   MAX_INT       : constant :=   (2**31)-1;
   MAX_DIGITS    : constant :=  9;
   MAX_MANTISSA : constant :=  31;
   FINE_DELTA    : constant :=  2.0**(-31);
   TICK          : constant :=  0.01;

-- Other System-Dependent Declarations

   subtype PRIORITY is INTEGER range 1..15;
--
-- The following exceptions are provided as a "convention"
-- whereby the Ada program can be compiled with all implicit
-- checks suppressed (i.e., pragma SUPPRESS or equivalent),
-- explicit checks included as necessary, the appropriate
-- exception raised when required, and then the exception is
-- either handled or the Ada program terminates.
--
   ACCESS_CHECK              : exception;
   DISCRIMINANT_CHECK        : exception;
   INDEX_CHECK               : exception;
   LENGTH_CHECK              : exception;
   RANGE_CHECK               : exception;
   DIVISION_CHECK            : exception;
   OVERFLOW_CHECK            : exception;
   ELABORATION_CHECK         : exception;
   STORAGE_CHECK             : exception;
--
-- The following exceptions provide for (1) Ada programs that
-- contain unresolved subprogram calls and (2) VAX/VMS
-- system-level errors.
--
   UNRESOLVED_REFERENCE      : exception;
   SYSTEM_ERROR              : exception;

end SYSTEM;
```

6.5 Character Set

Ada compilations may be expressed using the following characters, in addition to the basic character set:

a. The lower case letters

b. ! $? @ [/] ^ ' { } ~

6.6 Declaration and Representation Restrictions

Declarations are described in Chapter 3 of the Ada Language Reference Manual. Representation specifications are described in Chapter 13 and discussed here.

In the following specifications, the capitalized word 'SIZE' indicates the number of bits used to represent an object the type under discussion. The upper case symbols D, L, and R correspond to those discussed in Section 3.5.9 of ANSI/MIL-STD-1815A.

6.6.1 Integer Types

Integer types are specified with constraints of the form:

```
range L..R
```

where:

```
R <= SYSTEM.MAX_INT & L >= SYSTEM.MIN_INT
```

For an integer type, length specifications of the form:

```
for T'SIZE use N;
```

may specify integer values N such that N is in 2..32,

```
R <= 2**(N-1)-1 & L >= -2**(N-1);
```

or else such that

```
R <= (2**N)-1 & L >= 0
```

and N is in 1..31.

For a stand-alone object of integer type, a default SIZE of 16 is used when:

```
R <= 2**15-1 & L >= 2**15
```

Otherwise a SIZE of 32 is used.

For components of integer types within packed composite objects, the smaller of the default stand-alone SIZE or the SIZE from a length specification will be used.

6.6.2 Floating Types

Floating types are specified with constraints of the form:

```
digits D
```

where D is an integer value in 1 through 9.

For floating point types, length specifications of the form:

```
for T'SIZE use N;
```

may specify integer values N = 32 when D <= 6,
or N = 64 when D <= 9.

When no length specification is provided, a size of 32 is used;
when D <= 6, 64 when D is 7 through 9.

6.6.3 Fixed Types

Fixed types are specified with constraints of the form:

```
delta D range L..R
```

where:

$$\frac{\max(\text{abs}(R), \text{abs}(L)) < 2^{**}31-1}{\text{actual delta}}$$

The actual delta defaults to the largest integral power of 2 less than or equal to the specified delta D. (This implies that fixed point values are stored right-aligned.)

For fixed point types, length specifications of the form:

```
for T'SIZE use N;
```

are permitted only when N = 32. Where no length specification is provided, a size of 32 is used.

Specifications of the form:

```
for T'SMALL use X;
```

where the actual delta X must be an integral power of 2 such that X <= D.

6.6.4 Enumeration Types

In the absence of a representation specification for an enumeration type T, the internal representation of T'FIRST = 0. The default SIZE for a stand-alone object of enumeration type T will be the smallest of the values 8, 16, or 32, such that the internal representation of T'FIRST and T'LAST both falls within the range:

```
-2**(T'SIZE-1) .. 2**(T'SIZE-1)-1.
```

For enumeration types, length specification of the form:

```
for T'SIZE use N;
```

and/or enumeration representations of the form:

```
for T use <aggregate>;
```

are permitted for N in 2..32, provided that the internal representations and the SIZE conform to the relationship specified above.

Or else for N in 1..31, is supported for enumeration types and provides an internal representation of:

```
T'FIRST>=0 .. T'LAST<=2**(T'SIZE)-1.
```

For components of enumeration types within packed composite objects, the smaller of the default stand-alone SIZE or the SIZE from a length specification will be used.

Enumeration representation on types derived from the predined type BOOLEAN will not be accepted, but length specifications will be accepted.

6.6.5 Access Types

For access type, T, length specifications of the form:

```
for T'SIZE use N;
```

will not effect the runtime implementation of T, therefore N = 32 is the only value permitted for SIZE, which is the value returned by the attribute.

For collection size specification of the form:

```
for T'STORAGE_SIZE use N;
```

for any value of N is permitted for STORAGE_SIZE (and that value will be returned by the attribute call). The collection size specification will effect the implementation of T and its collection at runtime by limiting the number of objects for type T that can be allocated.

6.6.6 Arrays and Records

For arrays and records, length specification of the form:

```
for T'SIZE use N;
```

may cause arrays and records to be packed, if required, to accommodate the length specification. If the SIZE specified is not large enough to contain all possible values of the component(s), a diagnostic message of severity ERROR is issued.

The PACK pragma may be used to minimize wasted space, if any, between components of arrays and records. The pragma causes the type representation to be chosen such that storage space requirements are minimized at the possible expense of data access time and code space.

For records, a component clause of the form:

```
at N [range i,.j]
```

specifies the allocation of components in a record. Bits are numbered 0..7 from the right and bit 8 starts at the right of the next higher-number byte. Each location specification must allow at least X bits of range, where X is large enough to hold any value of the subtype of the component being allocated. Otherwise, a diagnostic message of severity ERROR is generated.

For records, an alignment clause of the form:

```
at mod N
```

specify alignments of N bytes for 1 byte, 2 bytes (VAX "word") and 4 bytes (VAX "long_word").

If it is determinable at compilation time that the SIZE of a record or array type or subtype maybe outside the range of STANDARD.LONG_INTEGER, a diagnostic message of severity WARNING is generated. Declaration of an object of such a type or subtype would raise NUMERIC_ERROR when elaborated. Note that a discrimated record or array may never raise the NUMERIC_ERROR when elaborated based on the actual discriminant provided.

6.6.7 Other Length Specifications

Length Specifications are described in Section 13.2 of the Ada Reference Manual.

A length specification for a task type T, of the form:

```
for T'STORAGE_SIZE use N;
```

specifies the number of bytes to be allocated for the runtime stack of each task object of type T.

6.7 System Names

Refer to Section 13.7 of the Ada Language Reference Manual for a discussion of package SYSTEM.

The available system names are "AdaVAX", "Ada_L", and "Ada_M"; and the system name is choosen based on the target(s) supported, but it can not be changed. In the case of VAX/VMS, the system name is "AdaVAX".

6.8 Address Clauses

Refer to Section 13.5 of the Ada Language Reference Manual for a discussion of Address Clauses. Address clauses have no effect for the VAX/VMS target.

The Runtime Support Library (RSL) for the VAX/VMS target does not handle hardware interrupts. All hardware interrupts are handled by the VAX/VMS operating system. However, the VAX_VMS target uses asynchronous system traps (ASTs) in a manner similar to interrupt entries.

6.9 Unchecked Conversions

Refer to Section 13.10.2 of the Ada Language Reference Manual for a description of UNCHECKED_CONVERSION.

A program is erroneous if it performs UNCHECKED_CONVERSION when the source and target have different sizes.

6.10 Restrictions on the Main (Sub)Program

Refer to Section 10.1 of the Ada Language Reference Manual for a discussion of the main (sub)program. The subprogram designated as the main (sub)program cannot have parameters. The designation as the main (sub)program of a subprogram whose specification contains a formal_part results in a diagnostic of severity ERROR at link time.

The main (sub)program can be a function, but the return value will not be available upon completion of the main (sub)program's execution. The main (sub)program may not be an imported subprogram.

6.11 Input/Output

Refer to Chapter 14 of the Ada Language Reference Manual for a description of Ada Input/Output.

6.11.1 External Files and Ada Input/Output

An external file name for Ada Input/Output can be any valid path name. The "FORM" string parameter to the CREATE and OPEN procedures in packages DIRECT_IO, SEQUENTIAL_IO and TEXT_IO are saved as long as the file is open. This allows the user to retrieve its value via calls to the procedure FORM, but it has no other effect.

6.11.2 File Processing

Processing allowed on ALS/N files is determined by the access controls set by the owner of the file and by the physical characteristics of the underlying device. The following restrictions apply:

a. A user may open a file as an IN_FILE only if that user has read access to the node. A user may open a file as an OUT_FILE only if that user has write access to the node. Finally, a user may open a file as an INOUT_FILE only if that user has read and write access to the node.

b. The attempt to CREATE a file with the mode IN_FILE is not supported since there will be no data in the file to read.

c. Multiple OPENs are allowed to read from a file, but all OPENs to write require exclusive access to the file. The exception USE_ERROR is raised if this restriction is violated.

d. No positioning operations are allowed on files associated with a printer or hard-copy terminal. The exception USE_ERROR is raised if this restriction is violated.

6.11.3 Text Input/Output

The implementation-defined type COUNT that appears in Section 14.3.10 of the Ada LRM is defined as follows:

```
type COUNT is range 0..INTEGER'LAST;
```

The implementation-defined subtype FIELD that appears in Section 14.3.10 of the Ada LRM is defined as follows:

```
subtype FIELD is INTEGER range 0..INTEGER'LAST;
```

At the beginning of program execution, the STANDARD_INPUT file and the STANDARD_OUTPUT file are open, and associated with the ALS/N-supported standard input and output files. Additionally, if a program terminates before an open file is closed (except for STANDARD_INPUT and STANDARD_OUTPUT), then the last line which the user put to the file may be lost.

A program is erroneous if concurrently executing tasks attempt to perform overlapping GET and/or PUT operations on the same terminal. Because of the physical nature of DecWriters and Video terminals, the semantics of text layout as specified in Ada Reference Manual Section 14.3.2 (especially the concepts of current column number and current line) cannot be guaranteed when GET operations are interweaved with PUT operations. Programs which rely on the semantics of text layout under those circumstances are erroneous.

For TEXT_IO processing, the line length can be no longer than the maximum VAX/VMS record length minus one, i.e., 255 characters. An attempt to set the line length through SET_LINE_LENGTH to a length greater than 255 will result in USE_ERROR. An attempt to write over the record length boundary will raise USE_ERROR.

6.11.4 Sequential Input/Output

The following restrictions are imposed on the use of the package Sequential_IO:

a. A null file name parameter to the CREATE procedure (for opening a temporary file) is not appropriate, and raises the exception NAME_ERROR;

b. Writing a record on a file associated with a tape adds the record to the file such that the record just written becomes the last record of the file;

c. On a disk or tape, the DELETE procedure closes the file and sets its size to zero so that its data may no longer be acessed; and

d. The subprogram END_OF_FILE always returns FALSE for a character-oriented device and RESET performs no action on a character-oriented device.

6.11.5 Direct Input/Output

The implementation-defined type COUNT that appears in Section 14.2.5 of the Ada LRM is defined as follows:

```
type COUNT is range 0..INTEGER'LAST;
```

6.11.6 Low Level Input/Output

The package LOW_LEVEL_IO defines a standard interface to allow an application to interact directly with a physical device. LOW_LEVEL_IO provides a definition of data types for a physical device and data to be operated on, along with the standard procedures SEND_CONTROL and RECEIVE_CONTROL. The procedure SEND_CONTROL may be used to send control infromation to a physical device. RECEIVE_CONTROL may be used to monitor the execution of an input/output operation by requestion information from a physical device.

```
with SYSTEM;
package LOW_LEVEL_IO is

    type IO_BUFFER_ADDRESS is new SYSTEM.ADDRESS;
    type IO_BUFFER_COUNT is new INTEGER;
    type IO_TIME_OUT is new INTEGER;

    type IO_FUNCTION is (
         read_data,     -- read data
         write_data,    -- write data
         initialize,    -- initialize the device and
                        -- return the device_code
         cancel,        -- cancel IO request
         control);      -- return control information

    type DEVICE_TYPE is new LONG_INTEGER;
         DEVICE_NAME_LENGTH: constant INTEGER := 32;

    type IO_REQUEST_BLOCK is record
         REQUESTED_FUNCTION: IO_FUNCTION;
         DEVICE_NAME:        STRING(1..DEVICE_NAME_LENGTH);
         DEVICE:             DEVICE_TYPE;
         BUFFER_ADDRESS:     IO_BUFFER_ADDRESS;
         BUFFER_COUNT:       IO_BUFFER_COUNT;
         TIME_OUT:           IO_TIME_OUT;
    end record;
```

```
    type IO_RETURN_STATUS is (
         ss_normal,          -- normal completion
         ss_abort,           -- all "failure" status codes
         ss_accvio,
         ss_devoffline,
         ss_exquota,
         ss_illefc,
         ss_insfmem,
         ss_ivchan,
         ss_nopriv,
         ss_unasefc,
         ss_linkabort,
         ss_linkdiscon,
         ss_protocol,
         ss_connecfail,
         ss_filalracc,
         ss_invlogin,
         ss_indevnam,
         ss_linkexit,
         ss_nolinks,
         ss_nosuchnode,
         ss_reject,
         ss_remrsrc,
         ss_shut,
         ss_toomuchdata,
         ss_unreachable);


    type IO_STATUS_BLOCK is record
         BYTE_COUNT:          IO_BUFFER_COUNT;
         RETURNED_STATUS:     IO_RETURN_STATUS;
    end record;

    procedure SEND_CONTROL (DEVICE: in    DEVICE_TYPE;
                            DATA: in out IO_REQUEST_BLOCK);

    procedure RECEIVE_CONTROL (DEVICE: in    DEVICE_TYPE;
                               DATA: in out IO_STATUS_BLOCK);

end LOW_LEVEL_IO;
```

6.12 System Defined Exceptions

In addition to the exceptions defined in the Ada Language Reference Manual, this implementation pre-defines the exceptions shown in Table 6-1 below.

Name	Significance
ACCESS_CHECK	The ACCESS_CHECK exception has been raised explicitly within the program.
DISCRIMINANT_CHECK	DISCRIMINANT_CHECK exception has been raised explicitly within the program.
INDEX_CHECK	The INDEX_CHECK exception has been raised explicitly within the program.
LENGTH_CHECK	The LENGTH_CHECK exception has been raised explicitly within the program.
RANGE_CHECK	The RANGE_CHECK exception has been raised explicitly within the program.
DIVISION_CHECK	The DIVISION_CHECK exception has been raised explicitly within the program.
OVERFLOW_CHECK	The OVERFLOW_CHECK exception has been raised explicitly within the program.
ELABORATION_CHECK	ELABORATION_CHECK exception has been raised explicitly within the program.
STORAGE_CHECK	The STORAGE_CHECK exception has been raised explicitly within the program.
UNRESOLVED_REFERENCE	Attempted call to a routine not linked into the executable image.
SYSTEM_ERROR	Serious error detected in underlying VAX/VMS operating system.

Table 6-1 - System Defined Exceptions

6.13 Machine Code Insertions

The Ada language definition permits machine code insertions as described in Section 13.8 of the Ada Reference Manual. This section describes the implementation specific details for writing machine code insertions as provided by the predefined library package MACHINE_CODE.

6.13.1 Machine Features

This section describes specific machine language features which are needed to write code statements. These machine features include the DISP and ADDRESS attributes and the address mode specifiers. The address mode specifiers make it possible to describe both the address mode and register number of any operand as a single value by mapping these values directly onto the first byte of each operand. The following is an enumeration of all mode specifiers:

```
--
-- The first 64 are the short literal modes.
-- These mode specifiers signify (short literal mode, value)
-- combinations.  The values are in the range 0 to 63.
--

        L0,             L1,             L2,             L3,
        L4,             L5,             L6,             L7,
        L8,             L9,             L10,            L11,
        L12,            L13,            L14,            L15,


        L16,            L17,            L18,            L19,
        L20,            L21,            L22,            L23,
        L24,            L25,            L26,            L27,
        L28,            L29,            L30,            L31,


        L32,            L33,            L34,            L35,
        L36,            L37,            L38,            L39,
        L40,            L41,            L42,            L43,
        L44,            L45,            L46,            L47,


        L48,            L49,            L50,            L51,
        L52,            L53,            L54,            L55,
        L56,            L57,            L58,            L59,
        L60,            L61,            L62,            L63,
```

```
--
-- Next are the (index mode, register) combinations.
--
        X_R0,           X_R1,           X_R2,           X_R3,
        X_R4,           X_R5,           X_R6,           X_R7,
        X_R8,           X_R9,           X_R10,          X_R11,
        X_AP,           X_FP,           X_SP,           X_PC,
--
-- The following are the (register mode, register) combinations.
--
        R0,             R1,             R2,             R3,
        R4,             R5,             R6,             R7,
        R8,             R9,             R10,            R11,
        AP,             FP,             SP,             PC,
--
-- The following are the (indirect register mode, register)
-- combinations.
--
        IR0,            IR1,            IR2,            IR3,
        IR4,            IR5,            IR6,            IR7,
        IR8,            IR9,            IR10,           IR11,
        IAP,            IFP,            ISP,            IPC,
--
-- Next are the (autodecrement register mode, register)
-- combinations.
--
        DEC_R0,         DEC_R1,         DEC_R2,         DEC_R3,
        DEC_R4,         DEC_R5,         DEC_R6,         DEC_R7,
        DEC_R8,         DEC_R9,         DEC_R10,        DEC_R11,
        DEC_AP,         DEC_FP,         DEC_SP,         DEC_PC,
--
-- Next are the (autoincrement register mode, register)
-- combinations. IMD (immediate mode) is autoincrement
-- mode using the PC.
--
        R0_INC,         R1_INC,         R2_INC,         R3_INC,
        R4_INC,         R5_INC,         R6_INC,         R7_INC,
        R8_INC,         R9_INC,         R10_INC,        R11_INC,
        AP_INC,         FP_INC,         SP_INC,         IMD,
--
-- The following are the (autoincrement deferred mode, register)
-- combinations.  A (absolute address mode) is autoincrement
-- deferred using the PC.
--
        IR0_INC,        IR1_INC,        IR2_INC,        IR3_INC,
        IR4_INC,        IR5_INC,        IR6_INC,        IR7_INC,
        IR8_INC,        IR9_INC,        IR10_INC,       IR11_INC,
        IAP_INC,        IFP_INC,        ISP_INC,        A,
```

```
--
-- The following are the (byte displacement mode, register)
-- combinations. B_PC is byte relative mode for the PC.
--
        B_R0,           B_R1,           B_R2,           B_R3,
        B_R4,           B_R5,           B_R6,           B_R7,
        B_R8,           B_R9,           B_R10,          B_R11,
        B_AP,           B_FP,           B_SP,           B_PC,
--
-- Next are the (byte displacement deferred mode, register)
-- combinations.  IB_PC is byte relative deferred mode for
-- the PC.
--
        IB_R0,          IB_R1,          IB_R2,          IB_R3,
        IB_R4,          IB_R5,          IB_R6,          IB_R7,
        IB_R8,          IB_R9,          IB_R10,         IB_R11,
        IB_AP,          IB_FP,          IB_SP,          IB_PC,
--
-- The following are the (word displacement mode, register)
-- combinations.  W_PC is word relative mode for the PC.
--
        W_R0,           W_R1,           W_R2,           W_R3,
        W_R4,           W_R5,           W_R6,           W_R7,
        W_R8,           W_R9,           W_R10,          W_R11,
        W_AP,           W_FP,           W_SP,           W_PC,
--
-- The following are the (word displacement deferred mode,
-- register) combinations.  IW_PC is word relative deferred
-- mode for the PC.
--
        IW_R0,          IW_R1,          IW_R2,          IW_R3,
        IW_R4,          IW_R5,          IW_R6,          IW_R7,
        IW_R8,          IW_R9,          IW_R10,         IW_R11,
        IW_AP,          IW_FP,          IW_SP,          IW_PC,
--
-- Next are the (longword displacement mode, register)
-- combinations.  L_PC is longword relative mode.
--
        L_R0,           L_R1,           L_R2,           L_R3,
        L_R4,           L_R5,           L_R6,           L_R7,
        L_R8,           L_R9,           L_R10,          L_R11,
        L_AP,           L_FP,           L_SP,           L_PC,
--
-- The following are the (longword displacement deferred mode,
-- register) combinations.  IL_PC is longword relative deferred
-- mode.
--
        IL_R0,          IL_R1,          IL_R2,          IL_R3,
        IL_R4,          IL_R5,          IL_R6,          IL_R7,
        IL_R8,          IL_R9,          IL_R10,         IL_R11,
        IL_AP,          IL_FP,          IL_SP,          IL_PC);
```

6.13.2 ADDRESS and DISP Attributes

The following restriction applies to the use of the ADDRESS and DISP attributes:

a. All displacements and addresses (i.e., branch destinations, program counter addressing mode displacements, etc.) must be static expressions.

b. Since neither the ADDRESS nor DISP attributes return static values, they may not be used in code statements.

6.13.3 Restrictions on Assembler Constructs

These unsupported Assembler constructs within the MACHINE_CODE package are as follows:

a. The VAX/VMS assembler's capability to compute the length of immediate and literal data is not replicated in MACHINE_CODE. This means the user cannot supply a value without specifying the length of that value. This disallows the assembler operand general formats: D(R), G, G^G, #cons, #cons[Rx], D(R)[Rx], G[Rx], G^location[Rx], @D(R)[Rx], @G[Rx], @D(R), @G such that D and G are byte, word or long_word values. Operands must contain address mode specifiers which explicitly define the length of any immediate or literal values of that operand.

b. The radix of the assembler notation is decimal. To express a hexadecimal literal, the notation 16#literal# should be used instead of ^X.

c. To construct an octaword, quadword, g_float or h_float number, it is important for the user to remember that the component fields of the records which make up the long numeric types are signed. This means that the user must take care to be assured that the values for these components, although signed, are interpreted correctly by the architecture.

d. Edit instruction streams must be constructed through the use of the VAX data statements described in Section 6.12.3.

e. Compatability mode instruction streams must be constructed through the use of the VAX data statements described in Section 6.12.3, if still supported on the VAX computer being utilized as the target machine (i.e., VAX-11/780 and 785, but not the VAX-8600).

f. No error messages are generated if the PC is used as the register for operands taking a single register, if the SP or PC are used for operands taking two registers, or if the AP, FP, SP or PC is used for operands taking four registers.

g. No error message is generated if the PC is used in register deferred or autodecrement mode.

h. If any register other than the PC is used as both the simple_operand and as the index_reg for an operand (see Section 6.14.1.2 for definitions of simple_operand and index_reg), no error message is generated. An example of this case is the VAX Assembler operand (7)[7].

i. Generic opcode selection is not supported. This means the opcode which reflects the specified number of operands must be used. For example, for 2 operand word addition, ADDW2 must be used, not just ADDW.

j. The PC is not supplied as a default if no register is specified in an operand. The user must supply the mode specifier which is mapped onto the PC. Examples are IMD, A, B_PC, W_PC, etc.

6.14 Machine Instructions and Data

This section describes the syntactic details for writing code statements (machine code insertions) as provided for the VAX by the pre-defined package MACHINE_CODE. The format for writing code statements is detailed, as are descriptions of the values to be supplied in the code statements. Each value is described by the named association for that value, and its defined in the order in which it must appear in positional notation. The programmer should refer to the VAX-11 Architecture Handbook along with this section to insure that the machine instructions are correct from an architectural viewpoint.

To insure a proper interface between Ada and machine code insertions, the user must be aware of the calling conventions used by the Ada compiler.

6.14.1 VAX Instructions

The general format for VAX code statements where the opcode is a one byte opcode is

```
BYTE_OP_CODE (OP => opcode {,"opcode"_1 => operand
                            {,"opcode"_2 => operand
                             {,"opcode"_3 => operand
                              {,"opcode"_4 => operand
                               {,"opcode"_5 => operand
                                {,"opcode"_6 => operand}}}}}});
```

The general format for VAX code statements where the opcode is a two byte opcode is

```
WORD_OP_CODE (OP => opcode2 {,"opcode2"_1 => operand
                             {,"opcode2"_2 => operand
                              {,"opcode2"_3 => operand
                               {,"opcode2"_4 => operand
                                {,"opcode2"_5 => operand
                                 {,"opcode2"_6 => operand}}}}}});
```

where "opcode"_n and "opcode2"_n is the result of the concatenation of the VAX opcode, an underscore, and the position of the operand in the VAX instruction. The BYTE_OP_CODE and WORD_OP_CODE statements always require an opcode and may include from 1 to 6 operands. The opcode mnemonics are precisely the same as described in the previously referenced VAX-11 Architecture Handbook. The VAX address modes divide the operands into six general categories, namely: Short Literal Operand, Indexed Operand, Register Operand, Byte Displacement Operand, Word Displacement Operand, and Long Word Displacement Operand.

6.14.1.1 Short Literal Operands

The VAX/VMS assembler format for short literal operands is

```
S^#cons
```

where cons is an integer constant with a range from 0 to 63 (decimal).

The code statement format for short literal operands is

```
(OP => short_lit)
```

where short_lit is one of the enumerated values, range L0 to L63, of the address mode specifiers in Section 6.11.1.

The following are examples of how some VAX Assembler short literals would be expressed in code statements.

```
S^#7   becomes  (OP => L7)
S^#33  becomes  (OP => L33)
S^#60  becomes  (OP => L60)
```

(For explanations of named and unnamed component association, see Section 4.3 of the Ada Language Reference Manual.)

6.14.1.2 Indexed Operands

The VAX/VMS Assembler format for the indexed operands is, in general

```
simple_operand[Rx]
```

where a simple_operand is an operand of any address mode except register, literal, or index.

The general code statement format for indexed operands is

```
(index_reg, simple_operand) or
    (OP => index_reg, OPND => simple_operand)
```

where index_reg is one of the enumerated address mode specifiers, range X_R0 to X_SP, from Section 6.11.1. Simple_operand is an operand of any address mode except register, literal, or index.

For example, the following indexed assembler operands,

- (R8)[R7] becomes (X_R7, (OP => IR8))
- (R8)+[R7] becomes (X_R7, (OP => R8_INC))
- I^#600[R4] becomes (X_R4, (IMD,600))
- -(R4)[R3] becomes (X_R3, (OP => DEC_R4))
- B^4(R9)[R3] becomes (X_R3, (B_R9,4))
- W^800(R8)[R5] becomes (X_R5, (W_R8,800))
- L^34000(R8)[R4] becomes (X_R4,(L_R8,34000))
- B^10[R9] becomes (X_R9, (B_PC,10))
- W^130[R2] becomes (X_R2, (W_PC,130))
- L^35000[R6] becomes (X_R6, (L_PC,35000))
- @(R3)+[R5] becomes (X_R5, (OP => IR3_INC))
- @#1432[R5] becomes (X_R5, (A,1432))
- @B^4(R9)[R3] becomes (X_R3, IB_R9,4))
- @W^8(R8)[R5] becomes (X_R5, (IW_R8,8))
- @L^2(R8)[R4] becomes (X_R4, (IL_R8,2))
- @B^3[R1] becomes (X_R1, (IB_PC,3))
- @W^150[R2] becomes (X_R2, (IW_PC,150))
- @L^100000[R3] becomes (X_R3, (IL_PC,100000))

would be expressed in named notation as:

```
o   (OP => X_R7, OPND => (OP => IR7))

o   (OP => X_R7, OPND => (OP => R8_INC))

o   (OP => X_R4, OPND => (OP => IMD, W_IMD => 600))

o   (OP => X_R3, OPND => (OP => DEC_R4))

o   (OP => X_R3, OPND => (OP => B_R9, BYTE_DISP => 4))

o   (OP => X_R5, OPND => (OP => W_R8, WORD_DISP => 800))

o   (OP => X_R4, OPND => (OP => L_R8,
                          LONG_WORD_DISP => 34000))

o   (OP => X_R9, OPND => (OP => B_PC, BYTE_DISP => 10))

o   (OP => X_R2, OPND => (OP => W_PC, WORD_DISP => 130))

o   (OP => X_R6, OPND => (OP => L_PC,
                          LONG_WORD_DISP => 35000))

o   (OP => X_R5, OPND => (OP => IR3_INC))

o   (OP => X_R5, OPND => (OP => A, ADDR => 1432))

o   (OP => X_R3, OPND => (OP => IB_R9, BYTE_DISP => 4))

o   (OP => X_R5, OPND => (OP => IW_R8, WORD_DISP => 8))

o   (OP => X_R4, OPND => (OP => IL_R8,
                          LONG_WORD_DISP => 2))

o   (OP => X_R1, OPND => (OP => IB_PC, B_DISP => 3))

o   (OP => X_R2, OPND => (OP => IW_PC, WORD_DISP => 150))

o   (OP => X_R3, OPND => (OP => IL_PC,
                          LONG_WORD_DISP => 100000))
```

6.14.1.3 Register Operands

The VAX/VMS Assembler formats for register operands are

```
  Rn     -- Register mode
 (Rn)    -- Register deferred mode
-(Rn)    -- Autodecrement mode
 (Rn)+   -- Autoincrement mode
@(Rn)+   -- Autoincrement deferred mode
```

where Rn represents a register numbered from 0 to 15.

The general code statement format for register operands is

```
(OP => regmode_value)
```

where regmode_value represents one of the enumerated address mode specifier range R0 to PC, from Section 6.11.1.

The following are examples of how VAX/VMS Assembler register mode operands would be written as code statements

```
  R7      becomes    (OP => R7)
 (R8)     becomes    (OP => IR8)
-(R9)     becomes    (OP => DEC_R9)
 (R1)+    becomes    (OP => R1_INC)
@(R3)+    becomes    (OP => IR3_INC)
```

6.14.1.4 Byte Displacement Operands

The VAX/VMS Assembler syntax for the byte displacement operands is

```
  B^d(Rn)        -- Byte displacement mode
 @B^d(Rn)        -- Byte displacement deferred mode
```

where d is the displacement added to the contents of register Rn. If no register is specified, the program counter is assumed. The code statement general format for the byte displacement and byte displacement deferred modes is

```
(byte_disp_spec, value)

          or

(OP => byte_disp_spec, BYTE_DISP => value)
```

where byte_disp_spec is one of the enumerated address mode specifiers, range B_R0 to B_PC for byte displacement or IB_R0 to IB_PC for byte displacement deferred, from Section 6.11.1. Value is in the range -128 to 127.

The following are examples of how VAX/VMS Assembler byte displacement operands would be written in code statements.

```
B^4(R5)      becomes  (B_R5, 4)      or
                       (OP => B_R5,  BYTE_DISP => 4)
B^200(R5)    becomes  (B_R5, 200)     or
                       (OP => B_R5,  BYTE_DISP => 200)
B^33         becomes  (B_PC, 33)      or
                       (OP => B_PC,  BYTE_DISP => 33)
@B^4(R5)     becomes  (IB_R5, 4)      or
                       (OP => IB_R5, BYTE_DISP => 4)
@B^200(R5)   becomes  (IB_R5, 200)    or
                       (OP => IB_R5, BYTE_DISP => 200)
@B^33        becomes  (IB_PC, 33)     or
                       (OP => IB_PC, BYTE_DISP => 33)
```

6.14.1.5 Word Displacement Operands

The VAX/VMS Assembler syntax for the word displacement operands are

```
W^d(Rn)        -- Word displacement
@W^d(Rn)       -- Word displacement deferred
```

where d is the displacement to be added to the contents of register Rn. If no register is specified, the program counter is assumed. In code statements, word displacement operands are represented in general as

```
(word_disp_spec, value)

or

(OP => word_disp_spec, WORD_DISP => value)
```

where word_disp_spec is one of the enumerated address mode specifiers, range W_R0 to W_PC for word displacement mode or IW_R0 or IW_PC for word displacement deferred mode, from Section 6.11.1. Value is in the range -2**15 to 2**15 - 1.

The following are examples of how VAX/VMS Assembler word displacement operands would be written in code statements.

```
W^10(R5)     becomes (W_R5, 10)   or
                     (OP => W_R5, WORD_DISP => 10)
W^20         becomes (W_PC, 20)   or
                     (OP => W_PC  WORD_DISP => 20)
@W^128(R7)   becomes (W_R7, 128)  or
                     (OP => IW_R7 WORD_DISP => 128)
@W^324       becomes (W_PC, 324)  or
                     (OP => IW_PC WORD_DISP => 324)
```

6.14.1.6 Longword Displacement Operands

The VAX/VMS Assembler general formats for the longword displacement operands is

```
L^d(Rn)          -- Long_word displacement
@L^d(Rn)         -- Long_word displacement deferred
```

where d is the displacement to be added to the register represented by Rn. Longword displacement operands are represented in code statements by the general format

```
(lword_disp_spec, value)

or

(OP => lword_disp_spec, LONG_WORD_DISP => value)
```

where lword_disp_spec is one of the enumerated address mode specifiers, range L_R0 to L_PC for long word displacement mode or IL_R0 to IL_PC for longword displacement deferred mode, from Section 6.11.1. Value is in the range -2**31 to 2**31 - 1.

The following are examples of how VAX/VMS Assembler long_word displacement operands would be written in code statements.

```
L^1000(R7)   becomes  (L_R7, 1000) or
                       (OP => L_R7, LONG_WORD_DISP => 1000)
L^25000      becomes  (L_PC, 25000) or
                       (OP => L_PC, LONG_WORD_DISP => 25000)
@L^1000(R9)  becomes  (IL_R9, 1000) or
                       (OP => IL_R9, LONG_WORD_DISP => 1000)
@L^3500      becomes  (IL_PC, 3500) or
                       (OP => IL_PC, LONG_WORD_DISP => 3500)
```

6.14.2 The CASE Statement

The VAX case statements (mnemonics CASEB, CASEW, and CASEL) have the following general symbolic form

```
opcode selector.rx, base.rx, limit.rx,
                    displ[0].bw, .. , displ[limit].bw
```

where x is dependent upon the opcode as to whether the operand is of type BYTE, WORD, or LONG_WORD. Displ[0].bw, .. , displ[limit].bw is a list of displacements to which to branch. Case statements would be written as code statements as:

```
BYTE_OP_CODE(OP => case_opcode,"case_opcode"_1=> operand,
             "case_opcode"_2 => operand,
              "case_opcode"_3 => case_operand)
```

where case_opcode is one of CASEB, CASEW, or CASEL. The type of operand and case operand are as indicated in the opcode (BYTE, WORD, or LONG_WORD). A case_operand is a special case operand of the form:

```
case_operand => (case_limit_address_mode, (case_enum))
```

or

```
case_operand => (LIMIT => case_limit_address_mode,
    (CASES=>case_enum))
```

if case_limit_address_mode is one of the short literal address specifiers. If case_limit_address_mode is the mode specifier IMD, the case_operand takes the form:

```
case_operand => (IMD, (case_limit, (case_enum)))
```

or

```
case_operand => (LIMIT => IMD, CASE_LIST =>
    (LIMIT => case_limit, (CASES => case_enum)))
```

where case_operand is one of BYTE_CASE_OPERAND, WORD_CASE_OPERAND, or LONG_WORD_CASE_OPERAND. The case_limit_address_mode is one of the short literal mode specifiers or the mode specifier IMD. Case_enum is a list of branch addresses. The branch addresses must be of type WORD. The case_limit is a value of the type indicated by the case_opcode.

Some examples of case statements written as code statements are:

```
<<START>>     BYTE_OP_CODE(CASEB, (OP =>R3, (IMD, 5), (IMD
                  (2,(15,30,45)))));  -- Case statement using
                                      -- immediate mode.

S2            BYTE_OP_CODE(CASEW, (OP => (W_PC, 10)), (IMD, 100),
                  (L2,(10,20,30)));   -- Case statement using
                                      -- short literal mode.
```

6.14.3 VAX Data

Constant values such as absolute addresses or displacements may be entered into the code stream with any of these nine statements:

```
BYTE_VALUE'(byte)
WORD_VALUE'(word)
LONG_WORD_VALUE'(long_word)
QUADWORD_VALUE'(quadword)
OCTAWORD_VALUE'(octaword)
FLOAT_VALUE'(float)
LONG_FLOAT_VALUE'(long_float)
G_FLOAT_VALUE'(g_float)
H_FLOAT_VALUE'(h_float)
```

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

```
-- MACRO.DFS                          -- ACVC_VERSION_1.10

-- THIS FILE CONTAINS THE MACRO DEFINITIONS USED IN THE ACVC TESTS.
-- THESE DEFINITIONS ARE USED BY THE ACVC TEST PRE-PROCESSOR,
-- MACROSUB. MACROSUB WILL CALCULATE VALUES FOR THOSE MACRO SYMBOLS
-- WHOSE DEFINITIONS DEPEND ON THE VALUE OF MAX_IN_LEN (NAMELY, THE
-- VALUES OF THE MACRO SYMBOLS BIG_ID1, BIG_ID2, BIG_ID3, BIG_ID4,
-- BIG_STRING1, BIG_STRING2, MAX_STRING_LITERAL, BIG_INT_LIT,
-- BIG_REAL_LIT, MAX_LEN_INT_BASED_LITERAL, MAX_LEN_REAL_BASED_LITERAL,
-- AND BLANKS).  THEREFORE, ANY VALUES GIVEN IN THIS FILE FOR THOSE
-- MACRO SYMBOLS WILL BE IGNORED BY MACROSUB.

-- NOTE: THE MACROSUB PROGRAM EXPECTS THE FIRST MACRO IN THIS FILE TO
--       BE MAX_IN_LEN.

-- EACH DEFINITION IS ACCORDING TO THE FOLLOWING FORMAT:

-- A. A NUMBER OF LINES PRECEDED BY THE ADA COMMENT DELIMITER, --.
--    THE FIRST OF THESE LINES CONTAINS THE MACRO SYMBOL AS IT APPEARS
--    IN THE TEST FILES (WITH THE DOLLAR SIGN).  THE NEXT FEW "COMMENT"
--    LINES CONTAIN A DESCRIPTION OF THE VALUE TO BE SUBSTITUTED.
--    THE REMAINING "COMMENT" LINES, THE FIRST OF WHICH BEGINS WITH THE
--    WORDS "USED IN: " (NO QUOTES), CONTAIN A LIST OF THE TEST FILES
--    (WITHOUT THE .TST EXTENSION) IN WHICH THE MACRO SYMBOL APPEARS.
--    EACH TEST FILE NAME IS PRECEDED BY ONE OR MORE BLANKS.
-- B. THE IDENTIFIER (WITHOUT THE DOLLAR SIGN) OF THE MACRO SYMBOL,
--    FOLLOWED BY A SPACE OR TAB, FOLLOWED BY THE VALUE TO BE
--    SUBSTITUTED.  IN THE DISTRIBUTION FILE, A SAMPLE VALUE IS
--    PROVIDED; THIS VALUE MUST BE REPLACED BY A VALUE APPROPRIATE TO
--    THE IMPLEMENTATION.

-- DEFINITIONS ARE SEPARATED BY ONE OR MORE EMPTY LINES.
-- THE LIST OF DEFINITIONS BEGINS AFTER THE FOLLOWING EMPTY LINE.

-- $MAX_IN_LEN
-- AN INTEGER LITERAL GIVING THE MAXIMUM LENGTH PERMITTED BY THE
-- COMPILER FOR A LINE OF ADA SOURCE CODE (NOT INCLUDING AN END-OF-LINE
-- CHARACTER).
-- USED IN:  A26007A
MAX_IN_LEN      120

-- $BIG_ID1
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX_IN_LEN.
-- THE MACROSUB PROGRAM WILL SUPPLY AN IDENTIFIER IN WHICH THE
-- LAST CHARACTER IS '1' AND ALL OTHERS ARE 'A'.
-- USED IN:  C23003A  C23003B  C23003C  B23003D  B23003E  C23003G
--           C23003H  C23003I  C23003J  C35502D  C35502F
BIG_ID1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1

-- $BIG_ID2
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX_IN_LEN,
-- DIFFERING FROM $BIG_ID1 ONLY IN THE LAST CHARACTER.  THE MACROSUB
-- PROGRAM WILL USE '2' AS THE LAST CHARACTER.
-- USED IN:  C23003A  C23003B  C23003C  B23003F  C23003G  C23003H
--           C23003I  C23003J
BIG_ID2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA2

-- $BIG_ID3
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX_IN_LEN.
-- MACROSUB WILL USE '3' AS THE "MIDDLE" CHARACTER; ALL OTHERS ARE 'A'.
-- USED IN:  C23003A  C23003B  C23003C  C23003G  C23003H  C23003I
--           C23003J
BIG_ID3 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA3AAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
-- $BIG_ID4
-- AN IDENTIFIER IN WHICH THE NUMBER OF CHARACTERS IS $MAX_IN_LEN,
-- DIFFERING FROM $BIG_ID3 ONLY IN THE MIDDLE CHARACTER.  MACROSUB
-- WILL USE '4' AS THE MIDDLE CHARACTER.
-- USED IN:  C23003A  C23003B  C23003C  C23003G  C23003H  C23003I
--           C23003J
BIG_ID4 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4AAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-- $BIG_STRING1
-- A STRING LITERAL (WITH QUOTES) WHOSE CATENATION WITH $BIG_STRING2
-- ($BIG_STRING1 & $BIG_STRING2) PRODUCES THE IMAGE OF $BIG_ID1.
-- USED IN:  C35502D  C35502F
BIG_STRING1     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

-- $BIG_STRING2
-- A STRING LITERAL (WITH QUOTES) WHOSE CATENATION WITH $BIG_STRING1
-- ($BIG_STRING1 & $BIG_STRING2) PRODUCES THE IMAGE OF $BIG_ID1.
-- USED IN:  C35502D  C35502F
BIG_STRING2     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1"

-- $MAX_STRING_LITERAL
-- A STRING LITERAL CONSISTING OF $MAX_IN_LEN CHARACTERS (INCLUDING THE
-- QUOTE CHARACTERS).
-- USED IN:  A26007A
MAX_STRING_LITERAL "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

-- $NEG_BASED_INT
-- A BASED_INTEGER LITERAL (PREFERABLY BASE 8 OR 16) WHOSE HIGHEST ORDER
-- NON-ZERO BIT WOULD FALL IN THE SIGN BIT POSITION OF THE
-- REPRESENTATION FOR SYSTEM.MAX_INT, I.E., AN ATTEMPT TO WRITE A
-- NEGATIVE VALUED LITERAL SUCH AS -2 BY TAKING ADVANTAGE OF THE
-- BIT REPRESENTATION.
-- USED IN:  E24201A
NEG_BASED_INT   16#FFFFFFFE#

-- $BIG_INT_LIT
-- AN INTEGER LITERAL WHOSE VALUE IS 298, BUT WHICH HAS
-- ($MAX_IN_LEN - 3) LEADING ZEROES.
-- USED IN:  C24003A
BIG_INT_LIT     000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000298

-- $BIG_REAL_LIT
-- A UNIVERSAL_REAL LITERAL WHOSE VALUE IS 690.0, BUT WHICH HAS
-- ($MAX_IN_LEN - 5) LEADING ZEROES.
-- USED IN:  C24003B  C24003C
BIG_REAL_LIT    000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000690.0

-- $MAX_LEN_INT_BASED_LITERAL
-- A BASED INTEGER LITERAL (USING COLONS) WHOSE VALUE IS 2:11:, HAVING
-- ($MAX_IN_LEN - 5) ZEROES BETWEEN THE FIRST COLON AND THE FIRST 1.
-- USED IN:  C2A009A
MAX_LEN_INT_BASED_LITERAL      2:000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000011:

-- $MAX_LEN_REAL_BASED_LITERAL
-- A BASED REAL LITERAL (USING COLONS) WHOSE VALUE IS 16:F.E:, HAVING
-- ($MAX_IN_LEN - 7) ZEROES BETWEEN THE FIRST COLON AND THE F.
-- USED IN:  C2A009A
MAX_LEN_REAL_BASED_LITERAL     16:00000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000F.E:
```

```
-- $BLANKS
-- A SEQUENCE OF ($MAX_IN_LEN - 20) BLANKS.
-- USED IN:  B22001A  B22001B  B22001C  B22001D  B22001E  B22001F
--           B22001G  B22001I  B22001J  B22001K  B22001L  B22001M
--           B22001N
--      <       LIMITS OF SAMPLE SHOWN BY ANGLE BRACKETS
                                >
BLANKS


-- $MAX_DIGITS
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX_DIGITS.
-- USED IN:  B35701A  CD7102B
MAX_DIGITS      9

-- $NAME
-- THE NAME OF A PREDEFINED INTEGER TYPE OTHER THAN INTEGER,
-- SHORT_INTEGER, OR LONG_INTEGER.
-- (IMPLEMENTATIONS WHICH HAVE NO SUCH TYPES SHOULD USE AN UNDEFINED
-- IDENTIFIER SUCH AS   NO_SUCH_TYPE_AVAILABLE.)
-- USED IN:  AVAT007  C45231D  B86001X C7D101G
NAME    NO_SUCH_TYPE_AVAILABLE

-- $FLOAT_NAME
-- THE NAME OF A PREDEFINED FLOATING POINT TYPE OTHER THAN FLOAT,
-- SHORT_FLOAT, OR LONG_FLOAT.  (IMPLEMENTATIONS WHICH HAVE NO SUCH
-- TYPES SHOULD USE AN UNDEFINED IDENTIFIER SUCH AS NO_SUCH_TYPE.)
-- USED IN:  AVAT013  B86001Z
FLOAT_NAME      NO_SUCH_TYPE_AVAILABLE

-- $FIXED_NAME
-- THE NAME OF A PREDEFINED FIXED POINT TYPE OTHER THAN DURATION.
-- (IMPLEMENTATIONS WHICH HAVE NO SUCH TYPES SHOULD USE AN UNDEFINED
-- IDENTIFIER SUCH AS NO_SUCH_TYPE.)
-- USED IN:  AVAT015  B86001Y
FIXED_NAME      NO_SUCH_TYPE_AVAILABLE

-- $INTEGER_FIRST
-- AN INTEGER LITERAL, WITH SIGN, WHOSE VALUE IS INTEGER'FIRST.
-- THE LITERAL MUST NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING
-- BLANKS.
-- USED IN:  C35503F  B54B01B
INTEGER_FIRST   -32768

-- $INTEGER_LAST
-- AN INTEGER LITERAL WHOSE VALUE IS INTEGER'LAST.  THE LITERAL MUST
-- NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING BLANKS.
-- USED IN:  C35503F  C45232A  B45B01B
INTEGER_LAST    32767

-- $INTEGER_LAST_PLUS_1
-- AN INTEGER LITERAL WHOSE VALUE IS INTEGER'LAST + 1.
-- USED IN:  C45232A
INTEGER_LAST_PLUS_1     32768

-- $MIN_INT
-- AN INTEGER LITERAL, WITH SIGN, WHOSE VALUE IS SYSTEM.MIN_INT.
-- THE LITERAL MUST NOT CONTAIN UNDERSCORES OR LEADING OR TRAILING
-- BLANKS.
-- USED IN:  C35503D  C35503F  CD7101B
MIN_INT -2147483648

-- $MAX_INT
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX_INT.
-- THE LITERAL MUST NOT INCLUDE UNDERSCORES OR LEADING OR TRAILING
-- BLANKS.
```

```
--  USED IN:  C35503D  C35503F  C4A007A  CD7101B
MAX_INT 2147483647

-- $MAX_INT_PLUS_1
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX_INT + 1.
-- USED IN:  C45232A
MAX_INT_PLUS_1  2147483648

-- $LESS_THAN_DURATION
-- A REAL LITERAL (WITH SIGN) WHOSE VALUE (NOT SUBJECT TO
-- ROUND-OFF ERROR IF POSSIBLE) LIES BETWEEN DURATION'BASE'FIRST AND
-- DURATION'FIRST.  IF NO SUCH VALUES EXIST, USE A VALUE IN
-- DURATION'RANGE.
-- USED IN:  C96005B
LESS_THAN_DURATION      -75000.0

-- $GREATER_THAN_DURATION
-- A REAL LITERAL WHOSE VALUE (NOT SUBJECT TO ROUND-OFF ERROR
-- IF POSSIBLE) LIES BETWEEN DURATION'BASE'LAST AND DURATION'LAST.  IF
-- NO SUCH VALUES EXIST, USE A VALUE IN DURATION'RANGE.
-- USED IN:  C96005B
GREATER_THAN_DURATION   75000.0

-- $LESS_THAN_DURATION_BASE_FIRST
-- A REAL LITERAL (WITH SIGN) WHOSE VALUE IS LESS THAN
-- DURATION'BASE'FIRST.
-- USED IN:  C96005C
LESS_THAN_DURATION_BASE_FIRST   -131073.0

-- $GREATER_THAN_DURATION_BASE_LAST
-- A REAL LITERAL WHOSE VALUE IS GREATER THAN DURATION'BASE'LAST.
-- USED IN: C96005C
GREATER_THAN_DURATION_BASE_LAST 131073.0

-- $COUNT_LAST
-- AN INTEGER LITERAL WHOSE VALUE IS TEXT_IO.COUNT'LAST.
-- USED IN:  CE3002B
COUNT_LAST      32767

-- $FIELD_LAST
-- AN INTEGER LITERAL WHOSE VALUE IS TEXT_IO.FIELD'LAST.
-- USED IN:  CE3002C
FIELD_LAST      32767

-- $ILLEGAL_EXTERNAL_FILE_NAME1
-- AN ILLEGAL EXTERNAL FILE NAME (E.G., TOO LONG, CONTAINING INVALID
-- CHARACTERS, CONTAINING WILD-CARD CHARACTERS, OR SPECIFYING A
-- NONEXISTENT DIRECTORY).
-- USED IN:  CE2103A  CE2102C  CE2102H  CE2103B  CE3102B  CE3107A
ILLEGAL_EXTERNAL_FILE_NAME1     BADCHAR^@.~!

-- $ILLEGAL_EXTERNAL_FILE_NAME2
-- AN ILLEGAL EXTERNAL FILE NAME, DIFFERENT FROM $EXTERNAL_FILE_NAME1.
-- USED IN:  CE2102C  CE2102H  CE2103A  CE2103B
ILLEGAL_EXTERNAL_FILE_NAME2     MUCH_TOO_LONG_NAME_FOR_A_FILE_UNDER_VMS_SO_THE

-- $ACC_SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS THE MINIMUM NUMBER OF BITS
-- SUFFICIENT TO HOLD ANY VALUE OF AN ACCESS TYPE.
-- USED IN:  CD2A81A  CD2A81B  CD2A81C  CD2A81D  CD2A81E
--           CD2A81F  CD2A81G  CD2A83A  CD2A83B  CD2A83C  CD2A83E
--           CD2A83F  CD2A83G  ED2A86A  CD2A87A
ACC_SIZE        32

-- $TASK_SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS THE NUMBER OF BITS REQUIRED TO
```

```
-- HOLD A TASK OBJECT WHICH HAS A SINGLE ENTRY WITH ONE INOUT PARAMETER.
-- USED IN:  CD2A91A  CD2A91B  CD2A91C  CD2A91D  CD2A91E
TASK_SIZE       1688

-- $MIN_TASK_SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS THE NUMBER OF BITS REQUIRED TO
-- HOLD A TASK OBJECT WHICH HAS NO ENTRIES, NO DECLARATIONS, AND "NULL;"
-- AS THE ONLY STATEMENT IN ITS BODY.
-- USED IN:  CD2A95A
MIN_TASK_SIZE   1584

-- $NAME_LIST
-- A LIST OF THE ENUMERATION LITERALS IN THE TYPE SYSTEM.NAME, SEPARATED
-- BY COMMAS.
-- USED IN:  CD7003A
NAME_LIST       VAX_VMS, ANUYK44, ANAYK14, ANUYK43

-- $DEFAULT_SYS_NAME
-- THE VALUE OF THE CONSTANT SYSTEM.SYSTEM_NAME.
-- USED IN:  CD7004A  CD7004C  CD7004D
DEFAULT_SYS_NAME        VAX_VMS

-- $NEW_SYS_NAME
-- A VALUE OF THE TYPE SYSTEM.NAME, OTHER THAN $DEFAULT_SYS_NAME.  IF
-- THERE IS ONLY ONE VALUE OF THE TYPE, THEN USE THAT VALUE.
-- NOTE: IF THERE ARE MORE THAN TWO VALUES OF THE TYPE, THEN THE
--       PERTINENT TESTS ARE TO BE RUN ONCE FOR EACH ALTERNATIVE.
-- USED IN:  ED7004B1  CD7004C
NEW_SYS_NAME    ANUYK43

-- $DEFAULT_STOR_UNIT
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.STORAGE_UNIT.
-- USED IN:  CD7005B  ED7005D3M  CD7005E
DEFAULT_STOR_UNIT       8

-- $NEW_STOR_UNIT
-- AN INTEGER LITERAL WHOSE VALUE IS A PERMITTED ARGUMENT  OR
-- PRAGMA STORAGE_UNIT, OTHER THAN $DEFAULT_STOR_UNIT.  IF THERE
-- IS NO OTHER PERMITTED VALUE, THEN USE THE VALUE OF
-- $SYSTEM.STORAGE_UNIT.  IF THERE IS MORE THAN ONE ALTERNATIVE,
-- THEN THE PERTINENT TESTS SHOULD BE RUN ONCE FOR EACH ALTERNATIVE.
-- USED IN:  ED7005C1  ED7005D1  CD7005E
NEW_STOR_UNIT   8

-- $DEFAULT_MEM_SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MEMORY_SIZE.
-- USED IN:  CD7006B  ED7006D3M  CD7006E
DEFAULT_MEM_SIZE        1073741823

-- $NEW_MEM_SIZE
-- AN INTEGER LITERAL WHOSE VALUE IS A PERMITTED ARGUMENT FOR
-- PRAGMA MEMORY_SIZE, OTHER THAN $DEFAULT_MEM_SIZE.  IF THERE IS NO
-- OTHER VALUE, THEN USE $DEFAULT_MEM_SIZE.  IF THERE IS MORE THAN
-- ONE ALTERNATIVE, THEN THE PERTINENT TESTS SHOULD BE RUN ONCE FOR
-- EACH ALTERNATIVE.  IF THE NUMBER OF PERMITTED VALUES IS LARGE, THEN
-- SEVERAL VALUES SHOULD BE USED, COVERING A WIDE RANGE OF
-- POSSIBILITIES.
-- USED IN:  ED7006C1  ED7006D1  CD7006E
NEW_MEM_SIZE    1073741823

-- $LOW_PRIORITY
-- AN INTEGER LITERAL WHOSE VALUE IS THE LOWER BOUND OF THE RANGE
-- FOR THE SUBTYPE SYSTEM.PRIORITY.
-- USED IN:  CD7007C
LOW_PRIORITY    1
```

```
-- $HIGH_PRIORITY
-- AN INTEGER LITERAL WHOSE VALUE IS THE UPPER BOUND OF THE RANGE
-- FOR THE SUBTYPE SYSTEM.PRIORITY.
-- USED IN:  CD7007C
HIGH_PRIORITY   15

-- $MANTISSA_DOC
-- AN INTEGER LITERAL WHOSE VALUE IS SYSTEM.MAX_MANTISSA AS SPECIFIED
-- IN THE IMPLEMENTOR'S DOCUMENTATION.
-- USED IN:  CD7013B
MANTISSA_DOC    31

-- $DELTA_DOC
-- A REAL LITERAL WHOSE VALUE IS SYSTEM.FINE_DELTA AS SPECIFIED IN THE
-- IMPLEMENTOR'S DOCUMENTATION.
-- USED IN:  CD7013D
DELTA_DOC       0.000_000_000_465_661_287_307_739_257_812_5

-- $TICK
-- A REAL LITERAL WHOSE VALUE IS SYSTEM.TICK AS SPECIFIED IN THE
-- IMPLEMENTOR'S DOCUMENTATION.
-- USED IN:  CD7104B
TICK    0.01
```

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

A39005G
This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

B97102E
This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

C97116A
This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

BC3009B
This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

CD2A62D
This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]
These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD2A81G, CD2A83G, CD2A84M & N, & CD50110
These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

CD2B15C & CD7205C
These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

CD2D11B
This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

CD5007B
This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

ED7004B, ED7005C & D, ED7006C & D [5 tests]
These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

CD7105A
This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK -- particular instances of change may be less (line 29).

CD7203B, & CD7204B
These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD7205D
This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

CE2107I
This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

CE3111C
This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

CE3301A
This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

CE3411B
This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

E28005C
This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY

U.S. NAVY

Compiler:	AdaVAX, Version 3.0 (/NO_OPTIMIZE Option)
ACVC Version:	1.10

Option	Function
Listing Control Options:	
ATTRIBUTE	Produce a symbol attribute listing. (Produces an attribute cross-reference listing when both ATTRIBUTE and CROSS_REFERENCE are specified.) Default: NO_ATTRIBUTE
DIAGNOSTICS	Produce a diagnostic summary listing. Default: NO_DIAGNOSTICS
MACHINE_CODE	Produce a machine code listing if code is generated. Default: NO_MACHINE_CODE Code is generated when CONTAINER_GENERATION option is in effect and (1) there are no diagnostics of severity ERROR, SYSTEM or FATAL (2) NO_CODE_ON_WARNING option is in effect and there are no diagnostics of severity higher than NOTE.
NOTES	Include diagnostics of NOTE severity level in the source or diagnostic summary listings. Default: NO_NOTES
PRIVATE	Include listing of Ada statements in private part if source listing is produced, subject to requirements of SOURCE option. Default: PRIVATE
SOURCE	Produce listing of Ada source statements. Default: NO_SOURCE
SUMMARY	Produce a summary listing, always produced when there are errors in the compilation. Default: NO_SUMMARY
CROSS_REFERENCE	Produce a cross-reference listing. (Produces an attribute cross-reference listing when both ATTRIBUTE and CROSS_REFERENCE are specified.) Default: NO_CROSS_REFERENCE

Table 10-1a - Ada Compiler Options

Option	Function
Special Processing Options:	
CHECKS	Provide run-time error checking. NO_CHECKS suppresses all run-time error checking. Please refer to the Pragma SUPPRESS description for further information on on run-time error checking. Default: CHECKS
CODE_ON_WARNING	Generate code (and machine code listing, if requested) only when there are no diagnostics of a severity higher than WARNING. Default: CODE_ON_WARNING NO_CODE_ON_WARNING means no code is generated when there is a diagnostic of severity WARNING or higher (i.e., ERROR, SYSTEM, or FATAL).
CONTAINER_GENERATION	Produce a Container if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is produced even if there are no diagnostices. No code (or machine code listing, if requested), is generated if a Container is not produced because the NO_CONTAINER_GENERATION option is in effect. Default: CONTAINER_GENERATION
DEBUG	Generates debugger symbolic information and, as required, changes the code being generated. If NO_DEBUG is specified, the compiler output includes only that information needed to link, export, and execute the current unit. Default: DEBUG The NO_DEBUG is ignored for a unit that: o is a package or subprogram specification, o is a subprogram body for which there is no previous declaration, or o contains a body stub, pragma INLINE, generic declaration, or a generic body. A diagnostic of severity NOTE is issued when the option is ignored.

Table 10-1b - Ada Compiler Options (con't)

Option	Function
MEASURE	Generates code to monitor execution frequency at the subprogram level for the current unit. Default: NO_MEASURE
OPTIMIZE	Enable global optimizations in accordance withthe optimization pragmas specified in the source program. Default: NO_OPTIMIZE When NO_OPTIMIZE is in effect, no global optimizations are performed, regardless of pragmas specified. The OPTIMIZE option enables global optimization. The goals of global optimization may be influenced by the user through the Ada-defined OPTIMIZE pragma. If TIME is specified, the global optimizer concentrates on optimizing execution time. If SPACE is specified, the global optimizer concentrates on optimizing code size. If the user does not include pragma OPTIMIZE, the optimizations emphasize TIME over SPACE. IF NO_OPTIMIZE is in effect, no optimizations are performed, regardless of the pragma.
TRACE_BACK	Provide calling sequence traceback information when a program is aborted because of an unhandled exception. Default: TRACE_BACK

Table 10-1c - Ada Compiler Options (con't)